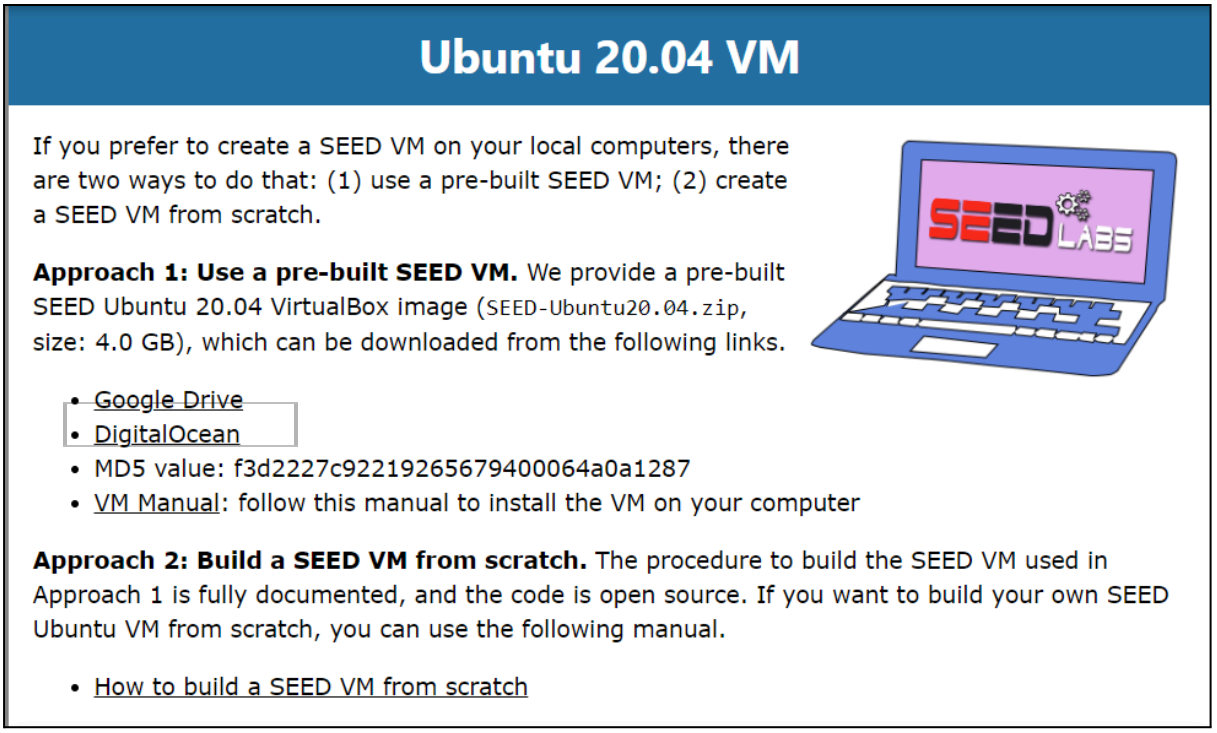
Network security - VPN Lab

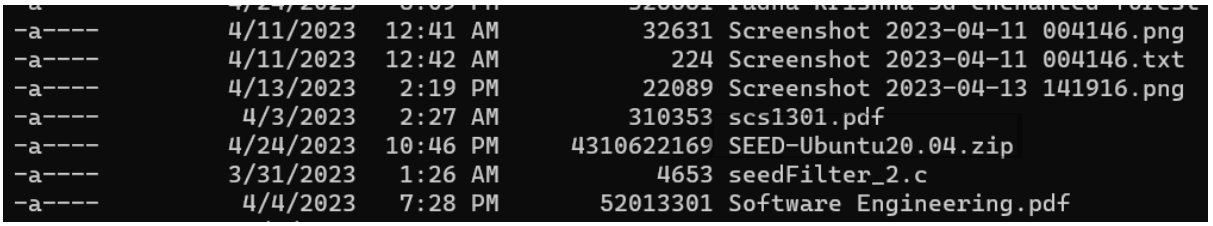
VPN Lab: The Container Version

Environment Setup Using Containers

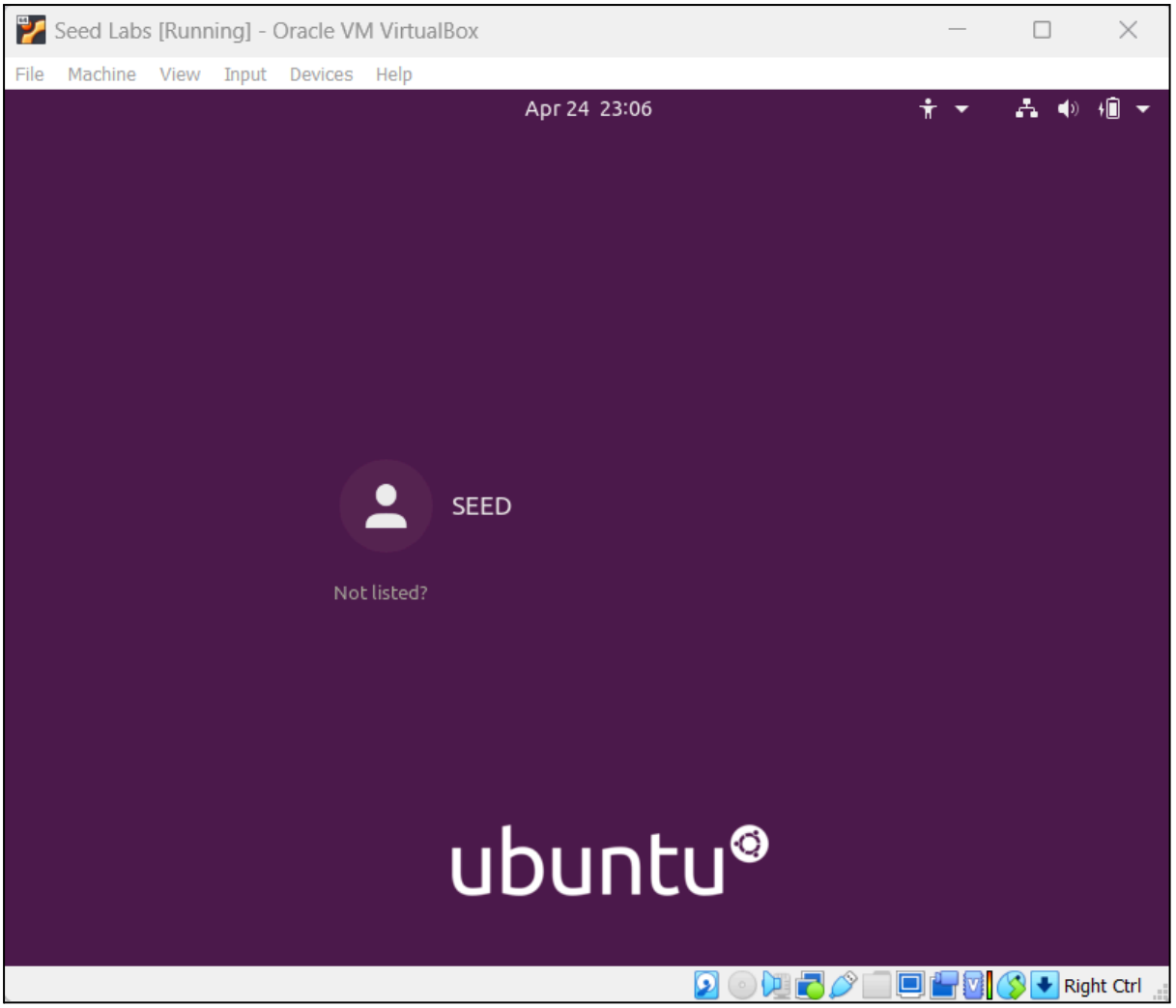
Container Setup and Commands

Download the Ubuntu 20.04 VM





Load the Virtual disk image into Virtual Box


Seed Labs [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Apr 24 23:06
SEED
Not listed?
ubuntu
Right Ctrl

Task 1: Network Setup

Inside VM - Download the *Labsetup.zip* and extract it

Overview

A Virtual Private Network (VPN) is a private network built on top of a public network, usually the Internet. Computers inside a VPN can communicate securely, just like if they were on a real private network that is physically isolated from outside, even though their traffic may go through a public network. VPN enables employees to securely access a company's intranet while traveling; it also allows companies to expand their private networks to places across the country and around the world.

The objective of this lab is to help students understand how VPN works. We focus on a specific type of VPN (the most common type), which is built on top of the transport layer. We will build a very simple VPN from the scratch, and use the process to illustrate how each piece of the VPN technology works. A real VPN program has two essential pieces, tunneling and encryption. This lab only focuses on the tunneling part, helping students understand the tunneling technology. The tunnel in this lab is not encrypted. There is another SEED lab that focuses on the encryption part.

Tasks (PDF)

- **VM version:** This lab has been tested on our SEED Ubuntu-20.04 VM
- **Lab setup files:**: Labsetup.zip
- **Manual:**: Docker manual

Extract

```
[04/24/23]seed@VM:~/Downloads$ ls -lrt
total 4
-rw-rw-r-- 1 seed seed 2155 Apr 24 23:11 Labsetup.zip
[04/24/23]seed@VM:~/Downloads$ unzip Labsetup.zip
Archive:  Labsetup.zip
   creating: Labsetup/
  inflating: Labsetup/docker-compose.yml
  inflating: Labsetup/docker-compose2.yml
   creating: Labsetup/volumes/
  inflating: Labsetup/volumes/tun.py
[04/24/23]seed@VM:~/Downloads$ ls -lrt
total 8
drwxr-xr-x 3 seed seed 4096 Dec  5  2020 Labsetup
-rw-rw-r-- 1 seed seed 2155 Apr 24 23:11 Labsetup.zip
[04/24/23]seed@VM:~/Downloads$ cd Labsetup
[04/24/23]seed@VM:~/.../Labsetup$ ls -lrt
total 12
drwxr-xr-x 2 seed seed 4096 Dec  5  2020 volumes
-rw-r--r-- 1 seed seed 2441 Dec  5  2020 docker-compose.yml
-rw-r--r-- 1 seed seed 3821 Dec  5  2020 docker-compose2.yml
[04/24/23]seed@VM:~/.../Labsetup$
```

Now, build the services in the VM using the file provided *docker-compose.yml* using command - '*dcbuild*'

dcbuild **helps to build or rebuild the services**

```
[04/24/23]seed@VM:~/.../Labsetup$ dcbuild
VPN_Client uses an image, skipping
Host1 uses an image, skipping
Host2 uses an image, skipping
Router uses an image, skipping
```

Now, startup the built services using docker command - '*dcup*'

```
[04/24/23]seed@VM:~/.../Labsetup$ dcup
Creating network "net-10.9.0.0" with the default driver
Creating network "net-192.168.60.0" with the default driver
Pulling VPN_Client (handsonsecurity/seed-ubuntu:large)...
large: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Extracting [==================================================>  ]  27.43MB/28.56MB
14428a6d4bcd: Download complete
2c2d948710f2: Download complete
b5e99359ad22: Downloading [==================================>                ]  35.07MB/52.67MB
3d2251ac1552: Download complete
1059cf087055: Download complete
b2afee800091: Download complete
c2ff2446bab7: Download complete
4c584b5784bd: Download complete
```

Now, verify the running and stopped containers by using a docker command - '*dockps*' (in new tab of the terminal, because the current tab has the services running up'

```
[04/24/23]seed@VM:~/.../Labsetup$ dockps
a9f3db837160  host-192.168.60.6
e2a4029627a9  host-192.168.60.5
9f1a06deac6b  client-10.9.0.5
0044a81ef78b  server-router
```

Now, login to '*client-10.9.0.5*', '*server-router*', '*host-192.168.60.5*', '*host-192.168.60.6*'

```
[04/24/23]seed@VM:~/.../Labsetup$ docksh client-10.9.0.5
root@9f1a06deac6b:/#
```

```
[04/24/23]seed@VM:~/.../Labsetup$ docksh server-router
root@0044a81ef78b:/#
```

```
[04/24/23]seed@VM:~/.../Labsetup$ docksh host-192.168.60.5
root@e2a4029627a9:/#
```

```
[04/24/23]seed@VM:~/.../Labsetup$ docksh host-192.168.60.6
root@a9f3db837160:/#
```

Changing the prompt to see the IPaddress clearly instead of some random characters used to protect the IPaddress

For '*client-10.9.0.5*'

```
root@9f1a06deac6b:/# echo $PS1
\u@\h:\w\$
root@9f1a06deac6b:/# export PS1="U-10.9.0.5:\w\n\$>"
U-10.9.0.5:/
$>
```

For '*server-router*'
There are 2 interfaces - '10.9.0.11' and '192.168.60.11'

```
[04/24/23]seed@VM:~/.../Labsetup$ docksh server-router
root@0044a81ef78b:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:0b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.11/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
14: eth1@if15: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:3c:0b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.60.11/24 brd 192.168.60.255 scope global eth1
       valid_lft forever preferred_lft forever
root@0044a81ef78b:/#
root@0044a81ef78b:/# export PS1="router-10.9.0.11-192.168.60.11;\w\n\$>"
router-10.9.0.11-192.168.60.11;/
$>
```

For '*host-192.168.60.5*'

```
[04/24/23]seed@VM:~/.../Labsetup$ docksh host-192.168.60.5
root@e2a4029627a9:/# export PS1="V-192.168.60.5:\w\n\$>"
V-192.168.60.5:/
$>
```

For '*host-192.168.60.6*'

```
[04/24/23]seed@VM:~/.../Labsetup$ docksh host-192.168.60.6
root@a9f3db837160:/# export PS1="V-192.168.60.6:\w\n\$>"
V-192.168.60.6:/
$>
```

The above steps were done just to avoid confusion while performing further tasks

Testing

a. Host U can communicate with VPN Server

```
U-10.9.0.5:/
$>ping 10.9.0.11 -c 4
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.080 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.155 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.185 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.085 ms

--- 10.9.0.11 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3065ms
rtt min/avg/max/mdev = 0.080/0.126/0.185/0.045 ms
```

Host U-10.9.0.5 can communicate with VPN Server 10.9.0.11 successfully

b. VPN Server can communicate with Host V

```
router-10.9.0.11-192.168.60.11;/
$>ping 192.168.60.5 -c 4
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=64 time=0.137 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.179 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.051 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=64 time=0.050 ms

--- 192.168.60.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3049ms
rtt min/avg/max/mdev = 0.050/0.104/0.179/0.055 ms
```

VPN Server 192.168.60.11 can communicate with Host V-192.168.60.5 successfully

c. Host U should not be able to communicate with Host V.

```
V-192.168.60.5:/
$>ping 10.9.0.5 -c 4
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
^C
--- 10.9.0.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3063ms

V-192.168.60.5:/
$>
```

d. Run tcpdump on the router, and sniff the traffic on each of the network. Show that you can capture packets.

From public/user network

```
router-10.9.0.11-192.168.60.11;/
$>tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
04:37:59.211165 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 14, seq 1, length 64
04:37:59.211194 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 14, seq 1, length 64
04:38:00.222793 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 14, seq 2, length 64
04:38:00.222855 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 14, seq 2, length 64
04:38:00.414954 IP6 fe80::1cb0:19ff:fec5:a878 > ff02::2: ICMP6, router solicitation, length 16
04:38:00.414987 IP6 fe80::42:47ff:fed4:291a > ff02::2: ICMP6, router solicitation, length 16
04:38:01.246815 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 14, seq 3, length 64
04:38:01.246883 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 14, seq 3, length 64
04:38:02.270764 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 14, seq 4, length 64
04:38:02.270828 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 14, seq 4, length 64
04:38:04.254893 ARP, Request who-has 10.9.0.5 tell 10.9.0.11, length 28
04:38:04.255417 ARP, Request who-has 10.9.0.11 tell 10.9.0.5, length 28
04:38:04.255445 ARP, Reply 10.9.0.11 is-at 02:42:0a:09:00:0b, length 28
04:38:04.255457 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, length 28
```

From Private network

```
router-10.9.0.11-192.168.60.11;/
$>tcpdump -i eth1 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
04:39:54.485291 IP 192.168.60.5 > 10.9.0.11: ICMP echo request, id 30, seq 1, length 64
04:39:54.485310 IP 10.9.0.11 > 192.168.60.5: ICMP echo reply, id 30, seq 1, length 64
04:39:55.486548 IP 192.168.60.5 > 10.9.0.11: ICMP echo request, id 30, seq 2, length 64
04:39:55.486564 IP 10.9.0.11 > 192.168.60.5: ICMP echo reply, id 30, seq 2, length 64
04:39:56.510609 IP 192.168.60.5 > 10.9.0.11: ICMP echo request, id 30, seq 3, length 64
04:39:56.510633 IP 10.9.0.11 > 192.168.60.5: ICMP echo reply, id 30, seq 3, length 64
04:39:57.534862 IP 192.168.60.5 > 10.9.0.11: ICMP echo request, id 30, seq 4, length 64
04:39:57.534893 IP 10.9.0.11 > 192.168.60.5: ICMP echo reply, id 30, seq 4, length 64
04:39:59.710528 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
04:39:59.710713 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
04:39:59.710723 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
04:39:59.710729 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
```

Task 2: Create and Configure TUN Interface

Task 2.a: Name of the Interface

```
U-10.9.0.5:/
$>cd volumes
U-10.9.0.5:/volumes
$>ls
tun.py
U-10.9.0.5:/volumes
$>chmod a+x tun.py
U-10.9.0.5:/volumes
$>./tun.py &
[1] 32
U-10.9.0.5:/volumes
$>Interface Name: tun0

U-10.9.0.5:/volumes
$>jobs
[1]+  Running                 ./tun.py &
U-10.9.0.5:/volumes
$>
```

Here, we gave execute permission to *tun.py* file and run it in the background as it takes a lot of time because of the infinite while loop.

```
$>ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

We can see the tun0 interface. We just created the interface, and still need to assign an IPaddress to it.

Kill the background process.

```
U-10.9.0.5:/volumes
$>jobs
[1]+  Running                 ./tun.py &
U-10.9.0.5:/volumes
$>kill %1
U-10.9.0.5:/volumes
$>jobs
[1]+  Terminated              ./tun.py
U-10.9.0.5:/volumes
```

Now as per the task, edit the *tun.py* and change the prefix of interface from *tun* to my last name - *gudibandi*

Open *tun.py ~/Downloads/Labsetup/volumes

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'gudibandi%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

while True:
    time.sleep(10)

```

After executing

```
U-10.9.0.5:/volumes
$>./tun.py &
[1] 42
U-10.9.0.5:/volumes
$>Interface Name: gudib0
```

```
U-10.9.0.5:/volumes
$>ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
5: gudib0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Task 2.b: Set up the TUN Interface

Configure the tun interface

Add the given 2 lines of code which configure the ip address to the tun interface to the *tun.py*

Open *tun.py ~/Downloads/Labsetup/volumes

```
1 #!/usr/bin/env python3
2
3 import fcntl
4 import struct
5 import os
6 import time
7 from scapy.all import *
8
9 TUNSETIFF = 0x400454ca
10 IFF_TUN   = 0x0001
11 IFF_TAP   = 0x0002
12 IFF_NO_PI = 0x1000
13
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'gudib%d', IFF_TUN | IFF_NO_PI)
17 ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19 # Get the interface name
20 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21 print("Interface Name: {}".format(ifname))
22
23 #configure the tun interface
24 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
25 os.system("ip link set dev {} up".format(ifname))
26
27 while True:
28    time.sleep(10)
29
```

After running the *tun.py* script, the ip address mentioned is attached to the tun interface

```
U-10.9.0.5:/volumes
$>ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
7: gudib0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global gudib0
       valid_lft forever preferred_lft forever
```

```
U-10.9.0.5:/volumes
$>ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
5: gudib0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Before running the configuration commands,

- The state of the was *UNKNOWN*, and now the state is *UP*
- The qdisc was in *fq_codel* i.e., Fair Queuing Controlled Delay, and now there is *noqueue*
- There is no inet address, now we can an inet address with scope global to *gudib0*
- There is no validity, and now the validity left forever is preferred to left forever.

Task 2.c: Read from the TUN Interface

Here, we are about to tun interface. Whatever coming out from the TUN interface is an IP packet. We can cast the data received from the interface into a Scapy IP object, so we can print out each field of the IP packet.

Replace the give while loop block in the *tun.py*

Open tun.py ~/Downloads/Labsetup/volumes Save

```
 3 import fcntl
 4 import struct
 5 import os
 6 import time
 7 from scapy.all import *
 8
 9 TUNSETIFF = 0x400454ca
10 IFF_TUN   = 0x0001
11 IFF_TAP   = 0x0002
12 IFF_NO_PI = 0x1000
13
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'gudib%d', IFF_TUN | IFF_NO_PI)
17 ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
18
19 # Get the interface name
20 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
21 print("Interface Name: {}".format(ifname))
22
23 #configure the tun interface
24 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
25 os.system("ip link set dev {} up".format(ifname))
26
27 while True:
28     # Get a packet from the tun
29     interfacepacket = os.read(tun, 2048)
30     if packet:
31         ip = IP(packet)
32         print(ip.summary())
```

On Host U, ping a host in the 192.168.53.0/24 network.

```
$>ping 192.168.53.5 -c 4
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw

--- 192.168.53.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3078ms
```

Observation

What are printed out by the tun.py program?
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw

This is a network packet that is using the Internet Protocol (IP) and the Internet Control Message Protocol (ICMP). The packet is being sent from the IP address 192.168.53.99 to the IP address 192.168.53.5.

The packet is an "echo-request" which is a type of ICMP message that is commonly used to test network connectivity. This message is often referred to as a "ping" request.

The "0" in this context typically represents the ICMP message identifier field.

The "Raw" field refers to the data payload of the packet, which can contain any type of data that the sender wants to include.

Overall, this packet is requesting an echo response from the destination IP address (192.168.53.5), which will indicate whether the destination host is reachable and responsive.

What has happened?
After sending 4 packets from the HOST-U to tun interface 192.168.53.5, there is 100% packet loss

Why?
Because the 192.168.53.5 is not present/available in the network

On Host U, ping a host in the internal network 192.168.60.0/24

```
$>ping 192.168.60.5 -c 4
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3058ms
```

Observation

Does tun.py print out anything?
No, there is no output

Why?
The packets transmitted to the internal network were transmitted via router, not through the tun interface. Because of that we have 100% loss on receiving the packets.

Task 2.d: Write to the TUN Interface

Attach the code as per instructed

```
27 while True:
28   # Get a packet from the tun interface
29   packet = os.read(tun, 2048)
30   if packet:
31     ip = IP(packet)
32     print(ip.summary())
33
34     #Send out a spoof packet using the tun interface
35     newip = IP(src='1.2.3.4', dst=ip.src)
36     newpkt = newip/ip.payload
37     os.write(tun, bytes(newpkt))
```

Now we need to add extra code to tun.py to meet the requirements

1. if this packet is an ICMP echo request packet, construct a corresponding echo reply packet and write it to the TUN interface under while loop

```
    # sniff and print out icmp echo request packet
     if ICMP in pkt and pkt[ICMP].type == 8:
         print("Original Packet.........")
         print("Source IP : ", pkt[IP].src)
         print("Destination IP :", pkt[IP].dst)

         # spoof an icmp echo reply packet
         # swap srcip and dstip
         ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
         icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
         data = pkt[Raw].load
         newpkt = ip/icmp/data

         print("Spoofed Packet.........")
         print("Source IP : ", newpkt[IP].src)
         print("Destination IP :", newpkt[IP].dst)

         send(newpkt, verbose=0)
         os.write(tun, bytes(newpkt))
```

```
while True:
  # Get a packet from the tun interface
  packet = os.read(tun, 2048)
  if packet:
    pkt = IP(packet)
    print(pkt.summary())

    #Send out a spoof packet using the tun interface
    # sniff and print out icmp echo request packet
    if ICMP in pkt and pkt[ICMP].type == 8:
        print("Original Packet.........")
        print("Source IP : ", pkt[IP].src)
        print("Destination IP :", pkt[IP].dst)

        # spoof an icmp echo reply packet
        # swap srcip and dstip
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data

        print("Spoofed Packet.........")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)

        send(newpkt, verbose=0)
        os.write(tun, bytes(newpkt))
```

Now, testing the code

After pinging to the tun interface ip 192.168.53.5 we received all the packets that were transmitted with 0% loss

```
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
Original Packet.........
Source IP :  192.168.53.99
Destination IP : 192.168.53.5
Spoofed Packet.........
Source IP :  192.168.53.5
Destination IP : 192.168.53.99
64 bytes from 192.168.53.5: icmp_seq=4 ttl=64 time=6.89 ms

--- 192.168.53.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 6.891/14.721/25.138/7.421 ms
U-10.9.0.5:/volumes
```

2. Sending arbitrary data instead of direct packet.

Here, pass the arbitrary data under the while loop

```
# sniff and print out icmp echo request packet
    if ICMP in pkt and pkt[ICMP].type == 8:
        print("Original Packet.........")
        print("Source IP : ", pkt[IP].src)
        print("Destination IP :", pkt[IP].dst)

        # spoof an icmp echo reply packet
        # swap srcip and dstip
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data

        print("Spoofed Packet.........")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)

        # send(newpkt, verbose=0)
        arbdata = b'some input'
        os.write(tun, bytes(arbdata))
```

```
  if packet:
    pkt = IP(packet)
    print(pkt.summary())

    #Send out a spoof packet using the tun interface
    # sniff and print out icmp echo request packet
    if ICMP in pkt and pkt[ICMP].type == 8:
        print("Original Packet.........")
        print("Source IP : ", pkt[IP].src)
        print("Destination IP :", pkt[IP].dst)

        # spoof an icmp echo reply packet
        # swap srcip and dstip
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data

        print("Spoofed Packet.........")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)

        #passing an arbitary data instead of a packet
        arbdata = b'Instead of writing an IP packet to the interface, write some arbitrary data to
the interface, and reportyour observation.'
        os.write(tun, bytes(arbdata))
```

Run *tun.py* and activate *tcpdump* to monitor the network

```
$>./tun.py &
[1] 116
U-10.9.0.5:/volumes
$>Interface Name: gudib0

U-10.9.0.5:/volumes
$>tcpdump -i eth0 -n 2> /dev/null &
[2] 124
U-10.9.0.5:/volumes
$>jobs
[1]-  Running                 ./tun.py &
[2]+  Running                 tcpdump -i eth0 -n 2> /dev/null &
```

Now test the code

```
$>jobs
[1]-  Running                 ./tun.py &
[3]+  Running                 tcpdump -i gudib0 -n 2> /dev/null &
U-10.9.0.5:/volumes
$>ping 192.168.53.5 -c 1
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
Original Packet.........
Source IP :  192.168.53.99
Destination IP : 192.168.53.5
Spoofed Packet.........
Source IP :  192.168.53.5
Destination IP : 192.168.53.99
00:54:26.119628 IP 192.168.53.99 > 192.168.53.5: ICMP echo request, id 127, seq 1, length 64
00:54:26.125024 IP truncated-ip - 29435 bytes missing! 114.105.116.105 > 110.103.32.97: ip-proto-102

--- 192.168.53.5 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

These 2 lines explains the background process done from the output

```
00:54:26.119628 IP 192.168.53.99 > 192.168.53.5: ICMP echo
request, id 127, seq 1, length 64
00:54:26.125024 IP truncated-ip - 29435 bytes missing!
114.105.116.105 > 110.103.32.97: ip-proto-102
```

00:54:26.119628 IP 192.168.53.99 > 192.168.53.5: ICMP echo request, id 127, seq 1, length 64

This packet is requesting an echo response from the destination IP address (192.168.53.5), which will indicate whether the destination host is reachable and responsive. The identifier and sequence number fields are used to match the response packet to this specific request packet.

00:54:26.125024 IP truncated-ip - 29435 bytes missing! 114.105.116.105 > 110.103.32.97: ip-proto-102

This packet is truncated, meaning that it has been cut short or partially lost in transit. The message indicates that "29435 bytes" are missing, which suggests that a large portion of the original packet was lost i.e., the arbitrary data we passed.

Here we are missing so much data, to analyze this objective, we need to know the ip packet format. For example, where does this ip address come from and what is the destination.

The source and destination ip address was generated based on the data we passed. The source ip address is stored from the 12th byte of the header. Hence the characters from the 12th position in the arbitrary data passed is ‘riti’ where the respective ascii values are 113, 105, 116 and 105. Similarly for destination ip address, it is stored from the 16th byte of the header i.e., The characters are ‘ting’ where the respective ascii values are 110, 103, 32, 97.

Task 3: Send the IP Packet to VPN Server Through a Tunnel

Create a new file as *tun_server.py* and copy the given code from instructions

tun.py | tun_server.py

```
1 #!/usr/bin/env python3
2
3 from scapy.all import *
4
5 IP_A = '0.0.0.0'
6 PORT = 9090
7
8 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
9 sock.bind((IP_A, PORT))
10
11 while True:
12   data, (ip,port) = sock.recvfrom(2048)
13   print("{}:{} --> {}:{}".format(ip,port, IP_A, PORT))
14   pkt = IP(data)
15   print("  Inside: {} --> {}".format(pkt.src, pkt.dst))
```

Now create *tun_client.py*

tun.py | tun_server.py | tun_client.py

```
8
9 TUNSETIFF = 0x400454ca
10 IFF_TUN   = 0x0001
11 IFF_TAP   = 0x0002
12 IFF_NO_PI = 0x1000
13
14 # Create UDP socket
15 sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
16 SERVER_IP, SERVER_PORT = 10.9.0.11, 9090
17
18 # Create the tun interface
19 tun = os.open("/dev/net/tun", os.O_RDWR)
20 ifr = struct.pack('16sH', b'gudib%d', IFF_TUN | IFF_NO_PI)
21 ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
22
23 # Get the interface name
24 ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
25 print("Interface Name: {}".format(ifname))
26
27 #configure the tun interface
28 os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
29 os.system("ip link set dev {} up".format(ifname))
30
31
32 while True:
33   # Get a packet from the tun interface
34   packet = os.read(tun, 2048)
35   if packet:
36     # Send the packet via the tunnel
37     sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Give execute permissions to *tun_server.py* and *tun_client.py* files

```
U-10.9.0.5:/volumes
$>chmod a+x tun_server.py
U-10.9.0.5:/volumes
$>chmod a+x tun_client.py
U-10.9.0.5:/volumes
```

Now run *tun_client.py* on Host U

```
U-10.9.0.5:/volumes
$>./tun_client.py &
[1] 138
U-10.9.0.5:/volumes
$>Interface Name: gudib0
```

It is running successfully

Now run *tun_client.py* on router

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server.py
```

It is also running without any errors.

Testing

Ping to VPN - 192.168.53.5 from Host - U

```
U-10.9.0.5:/volumes
$>ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1018ms
```

On the client side, the connection is timed out and we receive packets with 100% loss, because we don't have this host.

Now check on the server side

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server.py
10.9.0.5:54113 --> 0.0.0.0:9090
  Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:54113 --> 0.0.0.0:9090
  Inside: 192.168.53.99 --> 192.168.53.5
```

On the server side, we have a UDP packet sent from the client to the VPN server.

Inside: there is a source ip address representing the packet sent from and a destination ip address representing the packet sent to.

10.9.0.5:54114 - UDP payload
0.0.0.0:9090 - Server

From above we can say that the tunnel is working.

Why?
The tunnel worked because the IP packet is kept inside the UDP payload and received by the server.

Access the hosts inside the private network

We need to add the ip route to achieve the goal

```
U-10.9.0.5:/volumes
$>ip route add 192.168.60.0/24 dev gudib0
U-10.9.0.5:/volumes
$>ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.5
192.168.53.0/24 dev gudib0 proto kernel scope link src 192.168.53.99
192.168.60.0/24 dev gudib0 scope link
U-10.9.0.5:/volumes
```

Testing
Ping packets to private network from client side i.e., Host-U

```
U-10.9.0.5:/volumes
$>ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1026ms
```

On the client side, the connection is timed out and we receive packets with 100% loss.

On server side

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server.py
10.9.0.5:54113 --> 0.0.0.0:9090
  Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:54113 --> 0.0.0.0:9090
  Inside: 192.168.53.99 --> 192.168.53.5
10.9.0.5:54113 --> 0.0.0.0:9090
  Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:54113 --> 0.0.0.0:9090
  Inside: 192.168.53.99 --> 192.168.60.5
```

On the server side, we can see the packets being transmitted through the VPN server.

We successfully sent the packets through the VPN tunnel, but didn't receive any packets.

Task 4: Set Up the VPN Server

Modify *tun_server.py*

tun.py | tun_server.py | tun_client

```python
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

#tun interface
TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'gudib%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

#configure the tun interface
os.system("ip addr add 192.168.53.11/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

#UDP server

IP_A = '0.0.0.0'
```

Modify *tun_client.py*

```
30
31 #routing
32 os.sytem("ip route add 192.168.60.0/24 dev {}".format(ifname)
33
```

Add routing to track at server side.

Testing

On internal host, run *tcpdump*

```
V-192.168.60.5:/
$>tcpdump -i eth0 -n 2>/dev/null
```

On router, run *tun_server.py*

```
$>./tun_server.py
Interface Name: gudib0
```

On Host U, run *tun_client.py*

```
$>./tun_client.py &
[1] 181
U-10.9.0.5:/volumes
$>Interface Name: gudib0
```

Ping from Host-U to Internal network

```
U-10.9.0.5:/volumes
$>ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1002ms
```

We can see that we successfully transmitted the packets, but no packets were received back.

On router

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server.py
Interface Name: gudib0
10.9.0.5:50857 --> 0.0.0.0:9090
  Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:50857 --> 0.0.0.0:9090
  Inside: 192.168.53.99 --> 192.168.60.5
```

We can see that the UDP packets are being successfully transmitted through the tunnel

On Internal host

```
V-192.168.60.5:/
$>tcpdump -i eth0 -n 2>/dev/null
03:06:52.963169 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 191, seq
1, length 64
03:06:52.963202 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 191, seq 1,
 length 64
03:06:53.963796 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 191, seq
2, length 64
03:06:53.963836 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 191, seq 2,
 length 64
03:06:57.994081 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
03:06:57.994589 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
03:06:57.994608 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
03:06:57.994613 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
03:08:37.577831 IP6 fe80::42:88ff:feaa:de81 > ff02::2: ICMP6, router solicitatio
n, length 16
```

We can see that there are 2 requests and 2 replies generated.
The 2 replies replied to 192.168.53.99 which is a VPN but we cannot see it on the server, that means they weren't able to go through the tunnel and reach the client because we hadn't set up the server routing yet. But we did prove that the packets go through the tunnel and go to the Host-V and Host-V also sent it back. Therefore, the ICMP packets arrived to Host-V

Task 5: Handling Traffic in Both Directions

Modify *tun_server.py*

```
#UDP server

IP_A = '0.0.0.0'
PORT = 9090

ip,port = '10.9.0.5', 1234

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))

while True:
  # this will block until at least one interface is ready
  ready,_,_ = select.select([sock,tun],[],[])

  for fd in ready:
    if fd is sock:
      data, (ip,port) = sock.recvfrom(2048)
      pkt = IP(data)
      print("From socket ==>: {} --> {}".format(pkt.src, pkt.dst))
      # ... (code needs to be added by students) ...
      os.write(tun, bytes(pkt))
    if fd is tun:
      packet = os.read(tun,2048)
      pkt = IP(packet)
      print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
      # ... (code needs to be added by students) ...
      sock.sendto(packet, (ip,port))
```

Modify *tun_client.py*

```
while True:
  # this will block until at least one interface is ready
  ready,_,_ = select.select([sock,tun],[],[])

  for fd in ready:
    if fd is sock:
      data, (ip,port) = sock.recvfrom(2048)
      pkt = IP(data)
      print("From socket ==>: {} --> {}".format(pkt.src, pkt.dst))
      # ... (code needs to be added by students) ...
      os.write(tun, bytes(pkt))
    if fd is tun:
      packet = os.read(tun,2048)
      pkt = IP(packet)
      print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
      # Send the packet via the tunnel
      sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Testing

Ping

On Host-U

```
U-10.9.0.5:/volumes
$>ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket ==>: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=8.24 ms
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket ==>: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=12.9 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 8.242/10.572/12.903/2.330 ms
```

On the client side, we successfully transmitted and also received the packets this time without any loss which means the tunnel worked as expected.

On router

```
$>./tun_server.py
Interface Name: gudib0
From UDP ==> 10.9.0.5:47660 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From UDP ==> 10.9.0.5:47660 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
```

On the server side, we can see that from the socket, the packet passes through the VPN client and reaches the internal host and in reply, from tun, the packet passess through the VPN.

On internal network

```
V-192.168.60.5:/
$>tcpdump -i eth0 -n 2>/dev/null
03:32:06.908963 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 213, seq
1, length 64
03:32:06.908979 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 213, seq 1,
 length 64
03:32:07.912277 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 213, seq
2, length 64
03:32:07.912294 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 213, seq 2,
 length 64
03:32:11.977900 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
03:32:11.978100 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
03:32:11.978110 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
03:32:11.978139 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
```

On the internal host, we see the requests and replies in between the VPN tunnel and the host.

On client side

Do the *tcpdump* to check the flow of the packets

```
U-10.9.0.5:/volumes
$>ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
From tun ==>: 192.168.53.99 --> 192.168.60.5
From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=7.12 ms
03:46:09.808997 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 226, seq
1, length 64
03:46:09.816087 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 226, seq 1,
 length 64
03:46:10.810949 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 226, seq
2, length 64
From tun ==>: 192.168.53.99 --> 192.168.60.5
From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.53.99
03:46:10.830593 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 226, seq 2,
 length 64
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=19.7 ms
```

We can see the requests and replies similar to the Internal network.

Telnet

On Host-U

```
From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
                                                From socket(IP) ==>: 192.168.60.5 --> 19
2.168.53.99
             03:50:56.520234 IP 192.168.60.5.23 > 192.168.53.99.47960: Flags [.],
ack 77, win 509, options [nop,nop,TS val 1644746504 ecr 3533050792], length 0
                                                                                 03:
50:56.566109 IP 192.168.53.99.47960 > 192.168.60.5.23: Flags [.], ack 72, win 50
2, options [nop,nop,TS val 3533050852 ecr 1644746493], length 0
                                                                     From tun ==>: 192
.168.53.99 --> 192.168.60.5
                                From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
                                                                         From socket(I
P) ==>: 192.168.60.5 --> 192.168.53.99
                                             03:50:56.687694 IP 192.168.60.5.23 > 192.1
68.53.99.47960: Flags [P.], seq 72:92, ack 77, win 509, options [nop,nop,TS val
1644746671 ecr 3533050852], length 20
                                            e2a4029627a9 login: 03:50:56.687705 IP 192.
168.53.99.47960 > 192.168.60.5.23: Flags [.], ack 92, win 502, options [nop,nop,
TS val 3533050974 ecr 1644746671], length 0
                                                 From tun ==>: 192.168.53.99 --> 192.1
68.60.5
```

We can see that the OS operating the telnet which is *u 20.04.1 LTS* i.e., Ubuntu 20.04.1 version

On router

```
From UDP ==> 10.9.0.5:47660 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.60.5 --> 192.168.53.99
From UDP ==> 10.9.0.5:47660 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From UDP ==> 10.9.0.5:47660 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.60.5 --> 192.168.53.99
From UDP ==> 10.9.0.5:47660 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From UDP ==> 10.9.0.5:47660 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From UDP ==> 10.9.0.5:47660 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From UDP ==> 10.9.0.5:47660 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
```

On Internal Network

```
 ack 52, win 502, options [nop,nop,TS val 3533050792 ecr 1644746488], length 3 [
telnet DO ECHO [|telnet]
03:50:56.519192 IP 192.168.60.5.23 > 192.168.53.99.47960: Flags [.], ack 77, win
 509, options [nop,nop,TS val 1644746504 ecr 3533050792], length 0
03:50:56.568047 IP 192.168.53.99.47960 > 192.168.60.5.23: Flags [.], ack 72, win
 502, options [nop,nop,TS val 3533050852 ecr 1644746493], length 0
03:50:56.686137 IP 192.168.60.5.23 > 192.168.53.99.47960: Flags [P.], seq 72:92,
 ack 77, win 509, options [nop,nop,TS val 1644746671 ecr 3533050852], length 20
03:50:56.689046 IP 192.168.53.99.47960 > 192.168.60.5.23: Flags [.], ack 92, win
 502, options [nop,nop,TS val 3533050974 ecr 1644746671], length 0
03:51:56.567364 IP 192.168.60.5.23 > 192.168.53.99.47960: Flags [P.], seq 92:129
, ack 77, win 509, options [nop,nop,TS val 1644806553 ecr 3533050974], length 37
03:51:56.569800 IP 192.168.60.5.23 > 192.168.53.99.47960: Flags [F.], seq 129, a
ck 77, win 509, options [nop,nop,TS val 1644806555 ecr 3533050974], length 0
03:51:56.571424 IP 192.168.53.99.47960 > 192.168.60.5.23: Flags [.], ack 129, wi
n 502, options [nop,nop,TS val 3533110855 ecr 1644806553], length 0
03:51:56.575770 IP 192.168.53.99.47960 > 192.168.60.5.23: Flags [F.], seq 77, ac
k 130, win 502, options [nop,nop,TS val 3533110860 ecr 1644806555], length 0
03:51:56.575786 IP 192.168.60.5.23 > 192.168.53.99.47960: Flags [.], ack 78, win
 509, options [nop,nop,TS val 1644806561 ecr 3533110860], length 0
03:52:01.609795 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
03:52:01.609828 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
```

From the above output, it was a bit noisy to understand, so again running the *tun_client.py* without noise.

On Host-U

```
U-10.9.0.5:/volumes
$>./tun_client.py &>/dev/null &
[1] 228
U-10.9.0.5:/volumes
$>
U-10.9.0.5:/volumes
$>telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
e2a4029627a9 login:
```

Now we can clearly see the host that is operating the telnet (Ubuntu 20.04.1 LTS) and login prompt.

Now, logging in to the internal network

```
e2a4029627a9 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@e2a4029627a9:~$
```

Successfully logged in to the internal host 192.168.60.5

```
seed@e2a4029627a9:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
12: eth0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
 group default
    link/ether 02:42:c0:a8:3c:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.60.5/24 brd 192.168.60.255 scope global eth0
       valid_lft forever preferred_lft forever
seed@e2a4029627a9:~$
```

Therefore, Telnet is successful.

Task 6: Tunnel-Breaking Experiment

Here the process is as follows:

- Login to the telnet
- Execute any command while the server is running
- Stop the server
- Try to type any command in the telnet connection window
- Check if anything we can enter
- Restart the server
- Go back to the telnet window and check if the command you want to type is presented or not.

Login to the telnet 192.168.60.5 and execute the date command

```
To restore this content, you can run the 'unminimize' command.
Last login: Wed Apr 26 04:01:43 UTC 2023 on pts/3
seed@e2a4029627a9:~$ date
Wed Apr 26 04:08:32 UTC 2023
seed@e2a4029627a9:~$
```

Successfully executed the *date* command while server is running

Stop the server

```
From tun ==>: 192.168.60.5 --> 192.168.53.99
From UDP ==> 10.9.0.5:53250 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
^Z[1]   Terminated                 ./tun_server.py

[2]+  Stopped                 ./tun_server.py
router-10.9.0.11-192.168.60.11:/volumes
$>
```

Server stopped

Now, try to type any command

```
To restore this content, you can run the 'unminimize' command.
Last login: Wed Apr 26 04:01:43 UTC 2023 on pts/3
seed@e2a4029627a9:~$ date
Wed Apr 26 04:08:32 UTC 2023
seed@e2a4029627a9:~$
```

Not able to type anything

Restart the server

```
router-10.9.0.11-192.168.60.11:/volumes
$>./tun_server.py
Interface Name: gudib0
From UDP ==> 10.9.0.5:53250 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From UDP ==> 10.9.0.5:53250 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From UDP ==> 10.9.0.5:53250 --> 0.0.0.0:9090
```

Check the window

```
To restore this content, you can run the 'unminimize' command.
Last login: Wed Apr 26 04:01:43 UTC 2023 on pts/3
seed@e2a4029627a9:~$ date
Wed Apr 26 04:08:32 UTC 2023
seed@e2a4029627a9:~$ date
```

I can see the date command that I tried to type before.

So, in the background the buffering happens since the server is stopped or interrupted. That is why when the server restarted again, the telnet got activated and took the input.

Task 7: Routing Experiment on Host V

Here we have to delete the default entry from the Host V

```
V-192.168.60.5:/
$>ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
```

The default entry is from 192.168.60.11 which is our server-router

```
$>ip route del default
V-192.168.60.5:/
$>ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
V-192.168.60.5:/
$>
```

The default entry is successfully deleted.

Now, we need to add an entry

```
V-192.168.60.5:/
$>ip route add 192.168.53.0/24 via 192.168.60.5
V-192.168.60.5:/
$>ip route
192.168.53.0/24 via 192.168.60.5 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
```

We added an entry that will be routed from 192.168.53.0/24 to 192.168.60.5

Task 8: VPN Between Private Networks

In this task we need to switch to another network setup. So, down the current network first and then build up the *docker-compose2* setup

Down the current network

```
[04/27/23]seed@VM:~/.../Labsetup$ dcdown
Stopping host-192.168.60.6 ... done
Stopping client-10.9.0.5   ... done
Stopping server-router     ... done
Stopping host-192.168.60.5 ... done
Removing host-192.168.60.6 ... done
Removing client-10.9.0.5   ... done
Removing server-router     ... done
Removing host-192.168.60.5 ... done
Removing network net-10.9.0.0
Removing network net-192.168.60.0
[04/27/23]seed@VM:~/.../Labsetup$
```

Successfully stopped the current network.

Now, build and up the *docker-compose2* setup

```
[04/27/23]seed@VM:~/.../Labsetup$ docker-compose -f docker-compose2.yml build
HostA uses an image, skipping
HostB uses an image, skipping
VPN_Client uses an image, skipping
Host1 uses an image, skipping
Host2 uses an image, skipping
Router uses an image, skipping
```

dcup

```
[04/27/23]seed@VM:~/.../Labsetup$ docker-compose -f docker-compose2.yml up
Creating network "net-192.168.50.0" with the default driver
Creating network "net-10.9.0.0" with the default driver
Creating network "net-192.168.60.0" with the default driver
Creating host-192.168.60.6 ... done
Creating host-192.168.60.5 ... done
Creating server-router     ... done
Creating host-192.168.50.5 ... done
Creating host-192.168.50.6 ... done
Creating client-10.9.0.5   ... done
Attaching to server-router, host-192.168.60.6, host-192.168.50.5, host-192.168.50.6, client-10.9.0.5, host-192.168.60.5
host-192.168.60.6 |  * Starting internet superserver inetd               [ OK ]
host-192.168.50.5 |  * Starting internet superserver inetd               [ OK ]
host-192.168.50.6 |  * Starting internet superserver inetd               [ OK ]
host-192.168.60.5 |  * Starting internet superserver inetd               [ OK ]
```

dockps

```
[04/27/23]seed@VM:~/.../Labsetup$ dockps
f864abf686a8  host-192.168.50.5
27ef512cf6c1  client-10.9.0.5
c4c354b71ad1  host-192.168.60.5
0bcf15c3588c  server-router
39ce39d14aea  host-192.168.50.6
aa07e445de9c  host-192.168.60.6
```

Login to *host-192.168.50.5* (Host-U)

```
[04/27/23]seed@VM:~/.../Labsetup$ docksh host-192.168.50.5
root@f864abf686a8:/# export PS1="U-192.168.50.5:\w\n\$>"
U-192.168.50.5:/
$>
```

Login to VPN Client

```
[04/27/23]seed@VM:~/.../Labsetup$ docksh client-10.9.0.5
root@27ef512cf6c1:/# export PS1="Client-192.168.50.12-10.9.0.12:\w\n\$>"
Client-192.168.50.12-10.9.0.12:/
$>
```

Login to server-router

```
[04/27/23]seed@VM:~/.../Labsetup$ docksh server-router
root@0bcf15c3588c:/# export PS1="Server-192.168.60.11-10.9.0.11:\w\n\$>"
Server-192.168.60.11-10.9.0.11:/
$>
```

Login to *host-192.168.60.5* (Host-V)

```
[04/27/23]seed@VM:~/.../Labsetup$ docksh host-192.168.60.5
root@c4c354b71ad1:/# export PS1="V-192.168.50.5:\w\n\$>"
V-192.168.50.5:/
$>
```

We need to implement the code, and set up the client VPN across the network where we can see a reply back if everything is set up correctly and then we break the VPN tunnel and again see whether there are ping requests or not.

Now we need to edit *tun_server.py* and *tun_client.py*
Add the following line in *tun_server.py*

```
#routing
os.system("ip route add 192.168.50.0/24 dev {}".format(ifname))
```

Testing

Ping

```
U-192.168.50.5:/
$>ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2040ms
```

The packets are not reaching the destination.

On client VPN run *tun_client.py* and on server-router run *tun_server.py*

```
Client-192.168.50.12-10.9.0.12:/
$>cd volumes/
Client-192.168.50.12-10.9.0.12:/volumes
$>./tun_client.py
Interface Name: gudib0
```

```
Server-192.168.60.11-10.9.0.11:/
$>cd volumes/
Server-192.168.60.11-10.9.0.11:/volumes
$>./tun_server.py
Interface Name: gudib0
```

Now ping again

```
U-192.168.50.5:/
$>ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=62 time=10.4 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=20.5 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 10.412/15.465/20.519/5.053 ms
```

Now, the packets successfully transmitted and received with 0% loss.

On client VPN side

```
Client-192.168.50.12-10.9.0.12:/volumes
$>./tun_client.py
Interface Name: gudib0
From tun ==>: 192.168.50.5 --> 192.168.60.5
From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.50.5
```

On server side

```
Server-192.168.60.11-10.9.0.11:/volumes
$>./tun_server.py
Interface Name: gudib0
From UDP ==> 10.9.0.12:51500 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From UDP ==> 10.9.0.12:51500 --> 0.0.0.0:9090
From socket(IP) ==>: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
```

On Internal host (tcpdump)

```
V-192.168.50.5:/
$>tcpdump -i eth0 -n 2>/dev/null
02:37:57.994063 IP 192.168.50.5 > 192.168.60.5: ICMP echo request, id 31, seq 1, length 64
02:37:57.994096 IP 192.168.60.5 > 192.168.50.5: ICMP echo reply, id 31, seq 1, length 64
02:37:58.989278 IP 192.168.50.5 > 192.168.60.5: ICMP echo request, id 31, seq 2, length 64
02:37:58.989294 IP 192.168.60.5 > 192.168.50.5: ICMP echo reply, id 31, seq 2, length 64
02:38:03.134775 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
02:38:03.134887 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
02:38:03.134891 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
02:38:03.134903 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
```

We can see 2 requests and 2 reply echos. So the tunnel is successfully set up.

So, now we can keep on pinging from Host-U to Host-V and interrupt the client VPN and check the results.

On Host-U

```
U-192.168.50.5:/
$>ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=62 time=10.2 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=19.2 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=62 time=14.2 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=62 time=18.7 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=62 time=22.4 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=62 time=6.94 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=62 time=9.49 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=62 time=9.01 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=62 time=16.0 ms
64 bytes from 192.168.60.5: icmp_seq=15 ttl=62 time=8.04 ms
64 bytes from 192.168.60.5: icmp_seq=16 ttl=62 time=2.70 ms
64 bytes from 192.168.60.5: icmp_seq=17 ttl=62 time=13.9 ms
64 bytes from 192.168.60.5: icmp_seq=18 ttl=62 time=4.37 ms
64 bytes from 192.168.60.5: icmp_seq=19 ttl=62 time=13.1 ms
64 bytes from 192.168.60.5: icmp_seq=20 ttl=62 time=20.8 ms
64 bytes from 192.168.60.5: icmp_seq=21 ttl=62 time=3.03 ms
^C
--- 192.168.60.5 ping statistics ---
21 packets transmitted, 16 received, 23.8095% packet loss, time 20141ms
rtt min/avg/max/mdev = 2.702/12.007/22.427/6.092 ms
```

On Client VPN

```
From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.50.5
^CTraceback (most recent call last):
  File "./tun_client.py", line 36, in <module>
    ready,_,_ = select.select([sock,tun],[],[])
KeyboardInterrupt

Client-192.168.50.12-10.9.0.12:/volumes
$>./tun_client.py
Interface Name: gudib0
From tun ==>: 192.168.50.5 --> 192.168.60.5
From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From UDP ==> 10.9.0.11:9090 --> 10.9.0.5
From socket(IP) ==>: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
```

I interrupted the client server in between the pinging from Host-U to Host-V, and the icmp_seq stopped after 9 and when I restarted the client VPN again the packets started traveling to and fro and the icmp_seq resumed from 15.

Out of 21 packets that are sent, in return only 16 packets received, the remaining 5 packets were lost when the client VPN got stopped.

Task 9: Experiment with the TAP Interface

Create *tap.py*

```
[04/27/23]seed@VM:~/.../volumes$ ls
tun_client.py  tun.py  tun_server.py
[04/27/23]seed@VM:~/.../volumes$ cp tun.py tap.py
[04/27/23]seed@VM:~/.../volumes$ gedit tap.py
[04/27/23]seed@VM:~/.../volumes$
```

Edit *tap.py*

```
13
14 # Create the tun interface
15 tun = os.open("/dev/net/tun", os.O_RDWR)
16 ifr = struct.pack('16sH', b'gudib%d', IFF_TAP | IFF_NO_PI)
17 ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
```

Just change from *IFF_TUN* to *IFF_TAP*
And replace the while loop as below

```
27 while True:
28   packet = os.read(tap,2048)
29   if packet:
30     ether = Ether(packet)
31     print(ether.summary())
32
```

Now run *tap.py* on client VPN

```
Client-192.168.50.12-10.9.0.12:/volumes
$>./tap.py &
[1] 41
Client-192.168.50.12-10.9.0.12:/volumes
$>Interface Name: gudib0
```

Testing
Ping from client VPN to VPN Server

```
Client-192.168.50.12-10.9.0.12:/volumes
$>ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
Ether / ARP who has 192.168.53.5 says 192.168.53.99
From 192.168.53.99 icmp_seq=1 Destination Host Unreachable
From 192.168.53.99 icmp_seq=2 Destination Host Unreachable

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 1023ms
pipe 2
```

Here, the summary is that ARP packets sent out to the tap interface *192.168.53.99* and *192.168.53.5* asked for physical address three times. The adress is not reachable as expected and also there is no host with ip as *192.168.53.5*. Here, we simply reads the *tap* interface.

Now, replace the while loop with the given code in *tap.py*

```
# generate a corresponding ARP reply and write it to the TAP interface.
while True:
  packet = os.read(tap, 2048)
  if packet:
    print("--------------------------------")
    ether = Ether(packet)
    print(ether.summary())

    # Send a spoofed ARP response
    FAKE_MAC = "aa:bb:cc:dd:ee:ff"
    if ARP in ether and ether[ARP].op == 1:
      arp = ether[ARP]
      newether = Ether(dst=ether.src, src=FAKE_MAC)
      newarp = ARP(psrc=arp.pdst, hwsrc=FAKE_MAC, pdst=arp.psrc,hwdst=ether.src, op=2)
      newpkt = newether/newarp

      print("***** Fake response: {}".format(newpkt.summary()))
      os.write(tun, bytes(newpkt))
```

Now again start ping

```
Client-192.168.50.12-10.9.0.12:/volumes
$>ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
--------------------------------
Ether / ARP who has 192.168.53.5 says 192.168.53.99
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.5
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
--------------------------------
Ether / IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1031ms
```

As expected, we got fake response of ARP generated packets with the physical address as required.

Now *arping* with the commands given

```
Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I gudib0 192.168.53.33
ARPING 192.168.53.33
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=0 time=9.304 usec
--------------------------------
Ether / ARP who has 192.168.53.33 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 192.168.53.33
42 bytes from aa:bb:cc:dd:ee:ff (192.168.53.33): index=1 time=2.443 msec
^C
--- 192.168.53.33 statistics ---
2 packets transmitted, 2 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.009/1.226/2.443/1.217 ms
```

With ARPING to *192.168.53.33* we successfully transmitted and recived the packets with 0% loss.

```
Client-192.168.50.12-10.9.0.12:/volumes
$>arping -I gudib0 1.2.3.4
ARPING 1.2.3.4
------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=0 time=11.367 usec
------------------------------
Ether / ARP who has 1.2.3.4 says 192.168.53.99 / Padding
***** Fake response: Ether / ARP is at aa:bb:cc:dd:ee:ff says 1.2.3.4
42 bytes from aa:bb:cc:dd:ee:ff (1.2.3.4): index=1 time=13.704 msec
^C
--- 1.2.3.4 statistics ---
2 packets transmitted, 2 packets received,   0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.011/6.858/13.704/6.846 ms
```

Same with the address *1.2.3.4*. The results are as expected
